# Generic Dual Redundant Controller Requirements Validation

Francis L. Schneider NASA Detailee
Jet Propulsion Laboratory/
NASA IVV Facility
100 University Drive, Fairmont WV 26554
sch@atlantis.ivv.nasa.gov
304-367-8304, 304-367-8211(FAX)
and
John R. Callahan
NASA&WVU Software Research Lab
NASA IV&V Facility
100 University Drive, Fairmont WV 26554
callahan@atlantis.ivv.nasa.gov
304-367-8235, 304-267-8211(FAX)
and
Steve Easterbrook
NASA/WVU Software Research Lab
NASA IV&V Facility
100 University Drive, Fairmont WV 26554
steve@atlantis.ivv.nasa.gov
304-367-8352, 304-267-8211(FAX)
31 March 1997

**Abstract**

This paper describes the preliminary work done to validate a Generic Dual Redundant System (GDRS) system. The GDRS consists of two synchronous distributed computing components. Its primary characteristic is that it should have fault tolerant behavior in the face of errors that are unpredictable at the outset. The system uses a variant of checkpointing and rollback schemes found in the literature. The scheme as used here is referred to as the Mark and Rollback process. A SPIN model of the form $m = \mathcal{P} \times A_{\neg f}$ was used. $\mathcal{P}$ is the finite state program representing the model and $A_{\neg f}$ is the composite Büchi automaton representing multiple requirements corresponding to their linear temporal logic representation $f$ such that $f = \wedge_{i=1}^{n} f_i$. The method used for validation consists of constructing a composite Büchi automaton built from individual Linear Temporal Logic assertions that capture system requirements. This automaton is then run against the system model in the usual way to validate the model [5]. A scheme is also suggested using the trail files produced by the SPIN system to validate the implementation. The results reported on here demonstrate that the composite $A_{\neg f}$ functions as required when driven by a highly simplified $\mathcal{P}$. This conclusion shows that it is possible to validate a mark (checkpointing) and rollback application of this type giving rise to significantly lower operational risk of failure and therefore increased reliability and availability.

**Keywords**

linear temporal logic, concurrent programming, communications protocol, checkpointing and rollback, mark and rollback, synchronous communication, validation

## 1 Introduction

This paper describes a work in progress to validate a Dual Redundant Spacecraft Controller. The development shows the construction of the validation model together with its accompanying Linear Temporal Logic formulae and their corresponding automata. Section 2 introduces the concept of the use of a modeling tool to validate a general system representable by a Finite State Machine (FSM). The SPIN/PROMELA modeling tool developed by Holzmann [5] is used throughout and the reader is assumed to be somewhat familiar with this system although that isn't essential for a reading of the paper. Section 3 shows that the GDRS behaves as a communications system - showing that the SPIN/PROMELA tool is particularly valuable as an analysis tool since it was designed to validate such systems. Section 4 summarizes state properties of the system to be modeled. Section 5 gives a high level description of the model. Section 6 describes the Statecharts language and introduces the Statecharts model for the GDRS. Section 7 develops the five major fault cases. Section 8 develops the Linear Temporal Logic for the validation model and the methodology developed in section 2 is applied to the GDRS giving an automaton that has the capability to validate the GDRS model. Section 9 shows the development of the PROMELA model that was used for the validation of Case 1. Section 10 gives the results of the modeling procedure. Section 11 discusses future work and section 12 gives conclusions.

## 2 Linear Temporal Logic Background

The SPIN/PROMELA modeling scheme derives much of its power from its ability to incorporate formal theorem proving elements into its search schemes. Büchi [1] discovered the fundamental relationship between finite automata and the second-order monadic calculi. This innovation made it possible to incorporate Linear Temporal Logic (LTL) assertions as components of computer modeling schemes.

A Büchi automaton is a nondeterministic Finite State Machine (FSM) $A = (\Sigma, \mathcal{S}, \mathcal{T}r, S_0, \mathcal{F})$. $\Sigma$ is the input alphabet, $\mathcal{S}$ is the set of states, $S_0$ the set of initial states, and $\mathcal{F}$ is the set of *accepting states.* $\mathcal{T}r \in \mathcal{S} \times \Sigma \times \mathcal{S}$ is the transition relation. If $(s, a, s') \in \mathcal{T}r$ then A can move from $s$ to $s'$ upon reading $a$. A *trace* or input word is an infinite sequence $\sigma = \sigma_1\ \sigma_2\ \sigma_3, \ldots$ $,\sigma_i \in \Sigma$, while a *run* r, over $\sigma$ is an infinite sequence $s_0 \xmapsto{\sigma_1} s_1 \xmapsto{\sigma_2} \ldots,$ $\in S_0$,$(s_i,\ \sigma_{i+1},\ s_{i+1}) \in \mathcal{T}r,\ i = 0, 1, \ldots$. A run r is said to be accepting iff there exists a state $g \in \mathcal{F}$ such that $g$ appears infinitely often in $r$. The *language* $\mathcal{L}(A)$ is the set of all traces $\sigma$ such that A has an accepting run over $\sigma$.

Let $f_i$ be an LTL assertion corresponding to a system requirement to be validated that generates automaton $A_i$. Given n Büchi automata of the form $A_i = (\Sigma_i, \mathcal{S}_i, \mathcal{T}r_i, S_{0i}, \mathcal{F}_i)$, they are closed under the operation of intersection. Their intersection $\bigcap_{i=1}^{n} A_i$ accordingly is a Büchi automaton, and it accepts the language $\bigcap_{i=1}^{n} \mathcal{L}(A_i)$. The LTL formula that generates this automaton has the form

$$f = \bigwedge_{i=1}^{i=n} f_i \tag{1}$$

Equation 1 allows multiple LTL formulae to be concatenated such that the resulting automaton will preserve the characteristics of the language accepted by each automaton were it to be implemented in isolation. This means that the set of all traces $\sigma$, that were recognized by each atomaton $A_i$ in isolation will also be recognized by composite the automaton $\bigcap_{i=1}^{n} \mathcal{L}(A_i)$.

By incorporating the Finite State Machine (FSM) representation of the formal properties to be validated by the model, the model can be routinely checked for the presence or the absence of the desired characteristics.

The SPIN/PROMELA system has an LTL translator that can produce the corresponding Büchi automaton from an input requirement expressed as an LTL formula.

The SPIN modeling system checks to see that finite state program $\mathcal{P}$ satisfies the temporal logic formula $f$. First, the global state graph of $\mathcal{P}$ is computed. Second, the Büchi automaton is constructed for $\neg f$: $A_{\neg f}$ . Third, the synchronous product $\mathcal{P} \times A_{\neg f}$ is computed. Finally, the validation run is performed on $\mathcal{P} \times A_{\neg f}$. For each state transition in $\mathcal{P}$, SPIN checks to see if a corresponding transition in $A_{\neg f}$ is possible. Once one of $A_{\neg f}$'s accepting states has been entered, it must be shown that that state is reachable from itself. When this happens, $A_{\neg f}$ will have been shown to have recognized a string $\sigma$ from the language generated from the original LTL formula $\neg f$. For efficiency, SPIN executes the 3 steps in 1 pass. At this point a trail file can be written showing the sequence of state transitions in $\mathcal{P}$ that gave rise to the accepting state in $A_{\neg f}$. This file can then be annotated and run as a test case against the implementation.

## 2.1 Requirements Completeness and Consistency Validation

A well formed FSM model exhibits requirements completeness and consistency properties. Requirements consistency demands that for each state one and no more than one exit (guard) condition be satisfied at the same time. If more than one exit condition is simultaneously true, the requirements are inconsistent. If at a certain execution point, none of the exit conditions from a state are true and the state is not a terminating state, the requirements are incomplete. In either case an adjustment would have to be made in the requirements or in the model as necessary to bring the system into conformance. Semantically, LTL formulae allow specific dynamic and static requirements to be validated.

Suppose no trace $\sigma$ from the language $\mathcal{L}(A_{\neg f})$ is accepted by $A_{\neg f}$ with respect to finite state program $\mathcal{P}$. For properly constructed $f_i$, this emptiness condition is evidence that the LTL assertion $f_i$ is not violated for the model. For example, consider the liveness condition $f_i = \Box(p \rightarrow \Diamond q)$ where $\Box$ and $\Diamond$ are the henceforth and eventually operators respectively. Then realization of the emptiness condition with respect to $A_{\neg f}$ is a proof of correctness on the model when p is known to be true at some execution point. Since safety properties represent conditions that should never occur, their emptiness conditions are more straight forward.

Having proved assertions about the system model does not constitute evidence that the implementation is error free even if the implementation is able to successfully execute the various edited trail files generated through the validation process as described above. Nevertheless, a much higher level of confidence can be achieved in this way. The use of the SPIN system in the manner described above can be used to bring the model and the implementation into agreement.

If undesirable behavior is found to be absent from the model but present in the implementation, the implementation can be brought into agreement with the model. Conversely, if an undesired behavior is found to be present in the model but not in the implementation, then the model can be brought into agreement with the implementation. In this way the implementation can be rapidly validated to a high degree of confidence. The benefits achieved here are due to the relatively fast execution speed of the model with its accompanying short search times.

## 3 GDRS Coordinates behavior as a Communications System

This section shows that the GDRS behaves as a communications system - This correspondence shows that the SPIN/PROMELA tool is particularly valuable as an analysis tool since it was designed to validate such systems. There is a one-to-one correspondence between the validation of a communications protocol and the validation of the mark and rollback process. Holzmann [5] has defined a communications protocol as a five component specification for how communication is to be carried out in an error free way among two or more separate elements. They are

1. The service to be provided by the protocol.
2. The assumptions about the environment in which the protocol is executed.
3. The vocabulary of messages used to implement the protocol
4. The encoding (format) of each message in the vocabulary

5. The procedure rules guarding the consistency of message exchanges

Element 5. is contained in the requirements and the design of the critical sequence application. And, it is the phase where the validation effort is concentrated.

The Mark and Rollback Process is isomorphic to a communications system. It has a communications protocol, exhibits layered construction, possesses liveness and safety properties, and uses synchronous communications between its dual redundant processor systems.

The following provides a brief and partial mapping of a sampling of selected elements from the Mark and Rollback Process to the five elements present in a communications protocol. It demonstrates how the above mentioned correspondence comes about.

1. The service provided by the protocol is to keep the prime and the online systems in synchronization. This is done so that the online string can achieve prime status as soon as possible should the prime system become inoperable.

2. The environmental assumptions are: the system interacts with an entity that provides information about faults.

3. The major vocabulary consists of the variables SFP, CS, and CM. SFP is the spacecraft fault protection flag. When this flag is set, the GDRS has experienced a fault that has not yet been repaired. The CS flag is set in the prime string and in the backup string when the critical sequence is being processed in each respective string. The CM flag is set to indicate that the critical sequence is active and to remind the strings that when an interfering fault is fixed, the suspended critical sequence needs to be restarted at the last valid aged mark point. Restart data germane to the mark and rollback process contains much more information that is used during certain types of resets to restore functionality.

4. The three protocol flags mentioned, use single bit encoding. When a flag is set its value is 1; when reset, its value is 0. Table 1 shows the encoding scheme.

5. The procedure rules are most complex to deal with, the hardest to specify, the most difficult to validate. Most of the validation work occurs here. Examples from the mark and rollback support application are that the protocol variables SFP, CM, and CS are to be broadcast once each second to the online string and actually also back to the prime string by the prime string to allow the prime string to check its own synchronization. During the running of a critical sequence, when the online string detects a

| Data Structure | Value | Meaning |
|---|---|---|
| SFP | 0001 | fault |
| | 0000 | cleared |
| CS | 0001 | CS executing |
| | 0000 | CS not executing |
| CM | 0001 | CS active or suspended |
| | 0000 | CS inactive and not suspended |

Table 1: Communication Flags

broadcast message of the form CM=1, CS=0, SFP=1, it must stop running its own critical sequence. The online string must remain in this state until it detects that the prime string has completed servicing the fault. It then resumes execution of the critical sequence at the aged mark point when it receives the message CM=1, CS=1, SFP=0.

This example shows a small subset of the actual elements and their procedure rules that belong in each category. The complete protocol specification is in excess of 80 pages.

The example shows that the two communicating strings function under the control of a communications protocol and that they can therefore be modeled as a communications system.

## 4 GDRS Properties

The GDRS is a real time reactive system. As such, it it acts upon and responds to its environment. Three major characteristics used to describe reactive systems are dynamic or liveness properties, invariance, and safety properties. Liveness properties express actions that are required to take place either now or at some future time.

To apply modeling tools to the GDRS finite state system, the number of states present in the system must be less than $10^9$. If exhaustive searching of the state space is required, a state space less than several hundred thousand states are required. The state space available to the entire system is of the order of $10^{40}$ states. Accordingly, the GDRS was partitioned into 5 equivalence classes by required functionality. This procedure allowed the 5 major fault categories to be validated separately. The resulting exercise estimated that none of the equivalence classes has more than several hundred thousand states. Accordingly, a model checking approach is feasible towards validating GDRS requirements.

## 5 GDRS High Level Model Description

The Generic Dual Redundant System consists of two hardware platforms running identical software for the purpose of maximizing system reliability and availability. The systems exchange information to synchronize software operation. One of the platforms has control of the system bus and is called the prime string. The other one is called the online string and it executes in synchronization or at most executes within one second of the prime string. Primary information exchanged between the two systems is by the synchronous (rendez-vous) communication of a table called the State Table Broadcast (STB). The STB is a 32 word table that is broadcast by the prime string to the online string and to itself once per second. This paper considers the case where each of the strings executes high priority programs called critical sequences that must be responsive to unknown faults. To this end, the strings use a variant of the checkpoint and rollback process found to work well in industrial applications [6]. The industrial systems usually do not use hard rollback points embedded in the code that correspond to completed transactions in the executable code. Rollback points are computed on the fly and recorded for possible use should a problem occur later. They could correspond to the successful completion of a data base read or write operation for example. Such a completion is referred to as a commit operation meaning that if a system crash occurs system operations could be rolled back to the location where the commit occurred and proceed on from here. As the next operation proceeds, it could be interrupted by a power outage in a remote unit; the execution point would then be rolled back to the last checkpoint; and execution restarted from there. Here the system works analogously except that the rollback points are now called mark points instead of checkpoints and they are hard coded into the executing program. Each mark point now delineates the completion of a sub-operation in the overall program or sequence that is being executed. Checkpointing systems are well suited to operation where the nature of a future fault is not known ahead of time. These systems are ideal for use in man-out-of-the loop systems such as those in spacecraft where the two way light time (TWLT) is too long to make human response a feasible or a practical option. The next section gives an example of how the GDRS system might be used in a man-out-of-the loop situation where autonomy in the face of faults is important.

### 5.1 Operational Example

Consider the case where the GDRS is used in the retrieval and return of a soil sample by a remote robot. The prime string would perform all activities in the critical sequence unless a fault precluded it from proceeding. In that case the online string would take over. The completion of the initial retrieval of the sample corresponds to an operation that need not be repeated. The code ending in the completion of this process would be delineated with a mark point. The next group of instructions corresponding to the next subsequence in the program might be the storage of the sample that was just retrieved. Having finished this subsequence, another mark point would be encountered in the critical sequence, and following this, the next set of instructions in the sequence would be executed and so on until the task was finally completed. If the storage of the soil sample were interrupted by the occurrence of a fault, the system would repair the fault; roll back control to the beginning of the last mark point; and continue execution from there. It would not be necessary to waste battery power or time to retrieve another sample since that was already achieved. In essence then, this process is important in situations where two conflicting goals must be met: (a) time and/or power considerations are of the essence and (b) system faults must be repaired before execution can proceed. The solution is to repair the fault but to only repeat any previously incomplete subset of the task at hand. This paper focuses on the validation of the mark and rollback process as it is supported by the GDRS.

### 5.2 String Operation and Mark Point Aging

The online string uses the STB received each second to keep itself executing its copy of the critical sequence in synchronization with the prime string. This is particularly important since fault protection operates only in the prime string. This is a fault containment requirement. It operates to keep the isolation and repair of faults to the prime string. During the interval the prime string is repairing a fault, the online string must stop executing the critical sequence and wait for the STB to tell it that the system fault protection manager has set the flag SFP to 0, thereby signaling it to proceed with the critical sequence. This system controls external elements that are mostly mechanical in nature. Accordingly, the software is in general always ahead of the hardware. For this reason, three full seconds of execution time are allowed to pass after a new mark point is encountered by the software before the encounter with the new mark point is recognized as a legitimate rollback point. This procedure is necessary to give any mechanical tasks a chance to be completed in the previous mark point interval. It also allows any faults to be properly logged to the previous mark point interval should they actually have occurred during the previous sequence but were not registered due to time delays incurred in reporting the fault.

To record the passage of time, each new mark point is aged each second by one second by moving it one level deeper in a three level buffer. Only mark points that have reached the bottom will be eligible for use in the rollback process. Figure 1 shows a high level snapshot of normal critical sequence operation in both strings.

### 5.3 Fault Handling

Two major scenarios are important here. First, certain types of faults are of such a nature that they can be repaired by the prime string. As explained above, when a fault occurs the three protocol flags (CS, CM, SFP) change state from (1, 1, 0) to (0, 1, 1). This information is broadcast to the online string on once per second. When the online string sees the SFP flag, it suspends operation of the executing critical sequence and waits for the prime string to repair the fault. Once the fault is repaired, the prime string can roll back to the last valid mark point and resume processing. The online string will see the new SFP flag in the STB message, rollback to the aged broadcast mark point and restart its copy of the critical sequence. Figure 2 illustrates this case.

If the prime string goes down, the online string will sense this condition; transition to prime status thereby taking control of the system bus; roll back to the last mark point; and resume execution of the critical sequence. Having done this, the CS flag would be set to 1 to indicate that the critical sequence is again running. Meanwhile the old prime string will try to repair the fault that brought it down initially. If the old prime string recovers it will go through a reboot phase. Noticing that there is already a prime string, it will then transition to online status. Figure 3 shows the processes involved in this scenario.

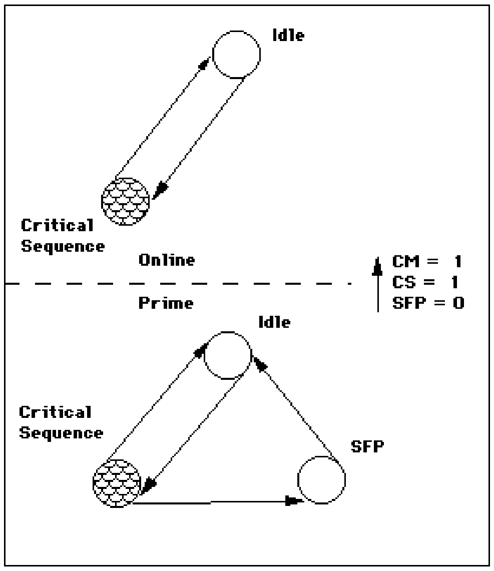

Figure 1: GDRS in Normal Dual String Critical Sequence Operation

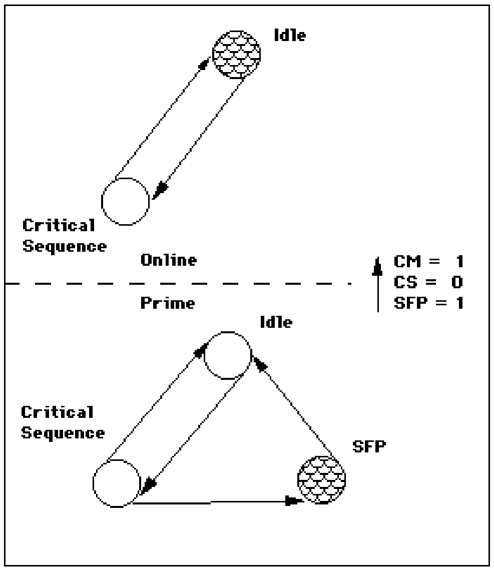

Figure 2: GDRS Dual String Operation with Prime String Fault in Progress

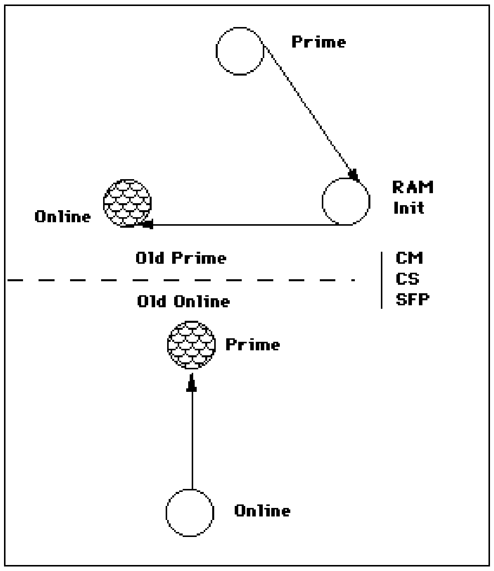

Figure 3: GDRS Platform String Swap

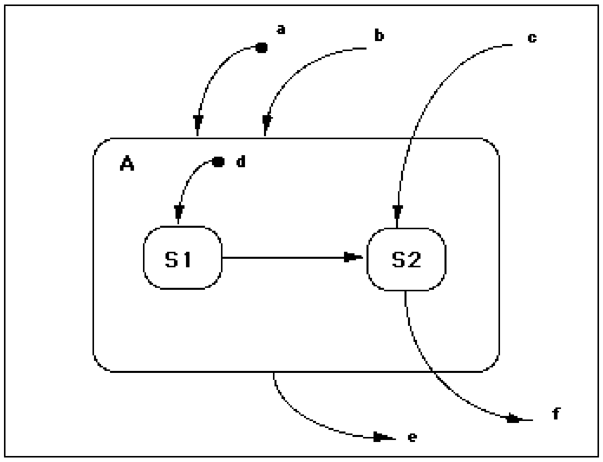

Figure 4: Harel Statechart Superstate Example

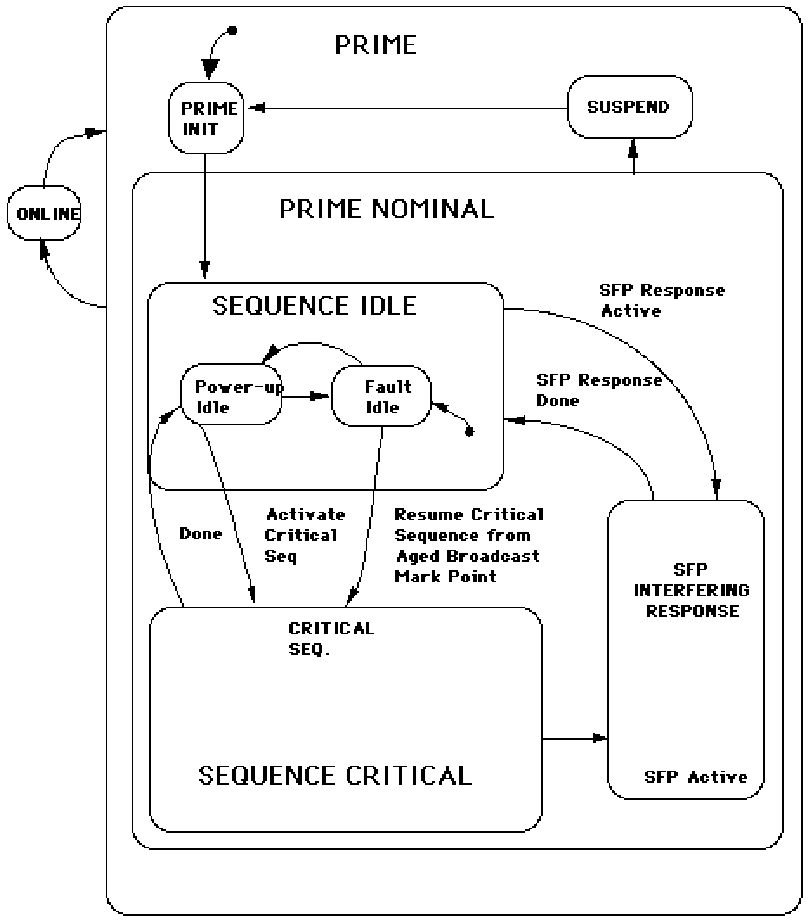

Figure 5: GDRS Prime String Harel State Chart

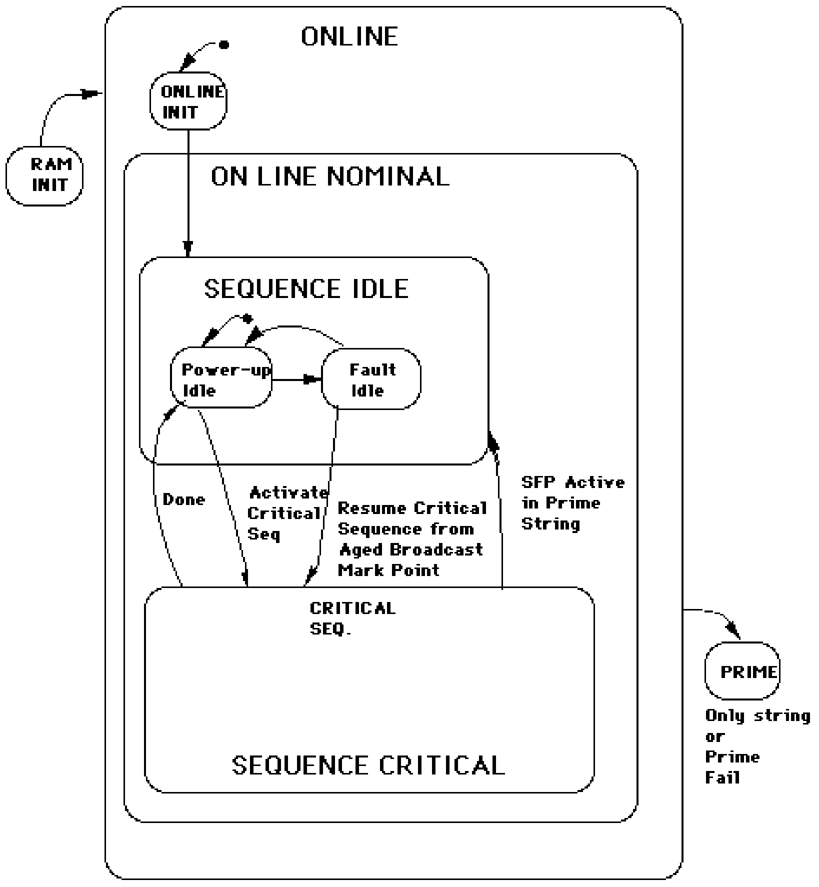

Figure 6: GDRS Online String Harel State Chart

## 6 The Statecharts Language

David Harel's Statecharts [2] [3] formalism is used to describe the GDRS system. Because we are using a highly simplified model keeping only characteristics germane to the mark and rollback process, we have selectively ignored many other threads that could be simultaneously executing. As a consequence, an understanding of the simplified system statechart as shown in to Figure 4 will be adequate for an understanding of the systems treated. The figure shows the notation used to describe the superstate. States as designated in a statechart are not the same concept as that discussed to determine the total number of allowable configurations for purposes of the validation discussed above. For purposes of understanding Statecharts, a state is a set of conditions and capabilities at a particular point in time or under a set of circumstances which characterize a things behavior. States may be grouped into super states. Accordingly, a super state is an abstraction of the states it encompasses, as A is to S1 and S2 (substates of A) in Figure 4. There are two ways to enter a super state. First, the transition to the superstate may end at the states border shown as transition a, the default entry state. Or if the default entry is not chosen a normal entry is shown as transition b. In either case a or b since entry is made at the boundary, the default state d must be specified within the state. A transition may also be made to a particular state within the super state as shown by transition c. Transitions out of the superstate can originate at the border. Transition e shows this case and it represents an exit from any of the states within the border. Transition f shows a valid transition out of superstate A from substate S2. Such transitions indicate an exit from any of the states that make up the superstate. Figures 5 and 6 show the simplified Harel Statecharts for the prime and online strings respectively. Key transitions are labeled with their corresponding actions. The validation model that is discussed in the rest of the paper is based upon these two statecharts. The discussion that follows also refers to the Statechart models of the GDRS shown in Figures 5 and 6.

## 7 Five Major Fault Cases

The GDRS system described here is a highly complex spacecraft controller containing many variables. As such it contains a large number of states that equate for purposes of modeling complexity to the total number of different values those variables in aggregate can take on. As pointed out in Section 4, this number easily gets to be too large a state space to search. Consequently, a key element in the practical validation of such systems is the recognition of what elements are significant and what elements can be regarded as extraneous to the validation. The procedure is then to model only the essentials germane to the mark and rollback model.

Recognizing that the five fault cases have the additional requirement imposed that the spacecraft system is not required to process multiple simultaneous faults, the five fault cases were partitioned into five separate cases. The validation becomes manageable as mentioned earlier. Should faults be detected during the repair of a fault condition, the system regards them as additional triggers of the current fault condition being processed and they are ignored. It is additionally necessary when considering each of the fault scenarios, to judiciously build a simplified system model so that validation times are brought as an initial goal into the feasible range of under an hour or so of search time for each fault condition. Further optimization can be realized by judiciously using model variables, data, and constants with the smallest allowable memory requirements. The five fault scenarios are (a) peripheral interfering fault (b) central interfering bus fault (c) prime central interfering fault (d) online fault and (e) undervoltage trip fault.

## 7.1 Case 1 Peripheral Interfering Fault

This fault is a spacecraft fault that is outside of the GDRS system per se. In this case the prime string is given the task of repairing the fault. The prime string would set the SFP flag to 1 to indicate a fault operation is in progress; stop the running critical sequence; set the CS flag to zero; and enter the SFP Active state to repair the fault. See Figure 5 for this transition. The STB would still be transmitted to the online string once per second. That is, since the fault is outside of the prime string, its ability to function has accordingly not been impaired. Having received the STB, the online string will cease running its copy of the critical sequence; set its own CS flag to 0; and transition to the Fault Idle state, waiting there until it receives an STB message indicating that it is all right to proceed with the critical sequence at the last valid mark point. Once the prime string has repaired the fault it sets its SFP flag to zero; enters the Fault Idle state in preparation for resuming the critical sequence. At this point it sets its CS flag to 1; rolls back to the last valid (aged) mark point; and resumes executing its copy of the critical sequence at this location. When the online string sees an STB message indicating that the SFP flag is is 0 it sets its CS flag to 1; enters the SEQUENCE CRITICAL state and resumes executing its copy of the critical sequence at the aged broadcast mark point. The states that apply and their transitions are illustrated in Figure 6.

### 7.2 Case 2 Online Fault

In this case the prime string keeps on executing the critical sequence since its operation isn't impaired. The online string attempts to repair itself and if successful attempts to resynchronize itself to the prime string so that it can be of further use should the prime string encounter difficulties in the future.

### 7.3 Case 3 Central Interfering Bus Fault

In this fault scenario the prime string is required to go into its SUSPEND state to repair the fault. In this state the STB ceases operation. Accordingly, when the online string senses the lack of two consecutive STB broadcasts it stops running its critical sequence; sets its CS flag to zero; and goes into its Fault Idle state awaiting the return of the STB indicating that it is all right to roll back to the last valid (aged) mark point and to proceed with the critical sequence as above. The online string senses the loss of two consecutive STBs but upon inspecting the cross-string-state vector sees that the prime string is processing a Central Interfering Bus Fault and is in fact not down. Here, when the prime string completes processing the fault, it must pass through the PRIME INIT state where it completes processing on its way to resuming the critical sequence through the sequence of PRIME INIT, Fault Idle and finally the SEQUENCE CRITICAL states as shown in Figure 5.

### 7.4 Case 4 Prime Central Interfering Fault

In this case the prime string itself is affected and it must go through a more severe reboot process than that discussed the Central Interfering Bus Fault case. Here, the online string similarly detects the loss of the STB for two consecutive seconds. Then checking the cross-string state vector, it sees that the prime string is in a prime central interfering fault response. Consequently, the online string transitions through the online state to become the new prime string and therefore takes control of the spacecraft bus. Figure 5 shows this transition. This configuration represents here the fact that the online string is only 1 interrupt away from becoming prime. Meanwhile to old prime string tries to recover. If it does so it goes through the reboot process in the RAM INIT state shown in Figure 6 and subsequently become the new online string if all goes well.

### 7.5 Case 5 Undervoltage Trip Fault

In this most challenging case a power supply failure causes one or both strings to partially or fully lose the ability to recover the critical sequence. Depending on the severity of the outage, none, one, or both strings may be forced to reset.

## 8 Linear Temporal Logic Formulae

This section develops the LTL formulae for several cases of interest. Each LTL formula will be incorporated into the resulting SPIN model as a ”never” clause. The resulting validation then proceeds as outlined in Section 2.

### 8.1 Rollback in Cases 1 Through 5

The LTL formulae developed here apply to rollback in all 5 cases. For concreteness, and since this paper focuses on validating Case 1, the result is presented with respect to a prime string peripheral interfering fault. The scenario examined here assumes a fault occurs external to the the prime and the online strings and that it is of such magnitude that it can be handled by the prime string alone without resorting to the backup string. As discussed above the state vector for the protocol flags (CS, CM, SFP) switches from (0,1,1) to $(1, 1, 0)$ upon repair of a peripheral fault. Following this operation, control in the prime string would roll back to the last valid mark point resuming execution at that point.

To validate that the correct rollback point is achieved, three separate conditions need to be responded to correctly.

1. If the last mark point was at the start of the program, roll back to the start regardless of how much time has expired since the program started running.

2. If the time following the last mark point was less than 3 seconds and the last mark point was not at the start of the program, roll back to the next previous mark point. That is, do not use the mark point that has not yet

been properly aged, even though it has been encountered in the execution of the current critical sequence.

3. If the time following the last mark point was greater than or equal to 3 seconds roll back to the last valid aged mark point.

All of these are liveness conditions; they specify an action that must take place now or in the future.

Symbolically, the LTL formulae representing these conditions have the form:

1. $\Box(p \rightarrow \Diamond q)$

meaning it is always the case that when p is true that now or in the future q will be true. Here p and q are Booleans such that the completed form for condition 1. is:

$p = (SFP = 1) \wedge (CS = 0) \wedge (CM = 1) \wedge (markpoint = start)$
$q = (pc = markpoint) \wedge (SFP = 0) \wedge (CS = 1) \wedge (CM = 1)$

where *markpoint* is the default mark point address of the beginning of the sequence; pc is the critical sequence machine program counter; and "start" is the address of the beginning of the critical sequence program. See Table 1 for the syntax and semantics of the flags CM, CS, SFP. Conditions 2. and 3. above similarly become:

2. $\Box(r \rightarrow \Diamond s)$ with r and s defined as

$r = (t < 3) \wedge (SFP = 1) \wedge (CS = 0) \wedge (CM = 1) \wedge (mp\_current \neq start)$
$s = (pc = m\_next\_previous) \wedge (SFP = 0) \wedge (CS = 1) \wedge (CM = 1)$

3. $\Box(u \rightarrow \Diamond v)$ with u and v defined as

$u = (t \geq 3) \wedge (SFP = 1) \wedge (CS = 0) \wedge (CM = 1) \wedge (mp\_current = mpt\_ge\_three\_sec)$
$v = (pc = mp\_current) \wedge (SFP = 0) \wedge (CS = 1) \wedge (CM = 1)$

where t is time in seconds since the last encountered mark point; here *mp_current* represents the current mark point and *mp_previous* represents the mark point preceding *mp_current*; each of these represents the case where less than three seconds have expired. *mpt_ge_three_sec* represents the mark point for the case where three or more seconds have expired since the last encounter of a mark point in the sequence.

Equation 1 allows 1. through 3. to be combined for purposes of generating the composite $f$:

$$f = \wedge_{i=1}^{3} f_i = \Box(p \rightarrow \Diamond q) \wedge (\Box(r \rightarrow \Diamond s) \wedge \Box(u \rightarrow \Diamond v) \qquad (2)$$

This formula can easily be translated into its corresponding automaton using the SPIN LTL translator. It can then be exhaustively validated with known input in isolation before being used with the model.

Since we want to demand that we know in each case that in fact a fault did occur, each element of equation 2 must be preceded by $\Diamond p$, $\Diamond r$ and $\Diamond u$ respectively giving:

$$f = \wedge_{i=1}^{3} f_i = \Diamond p \wedge \Box(p \rightarrow \Diamond q) \wedge \Diamond q \wedge (\Box(r \rightarrow \Diamond s) \wedge \Diamond u \wedge \Box(u \rightarrow \Diamond v) \quad (3)$$

Three analogous conditions could be written down for the online string using its copies of the 3 protocol variables. Each of these giving another three $f_i$ to be concatenated with the result in equation 2:

$f_4 = \Diamond h \wedge \Box(h \rightarrow \Diamond i)$
$f_5 = \Diamond j \wedge \Box(j \rightarrow \Diamond k)$
$f_6 = \Diamond l \wedge \Box(l \rightarrow \Diamond m)$

To demand that when a given type of rollback takes place in the prime system, that in fact the corresponding type of rollback takes place in the online system, the corresponding rollback conditions must be logically tied together. Equation 4 demonstrates one way to accomplish this for the case of rollback to the start of the sequence:

$$f = \Diamond p \wedge \Box(p \rightarrow \Diamond q) \wedge \Diamond i \qquad (4)$$

In this way the rollback could be validated in each string when (a) the prime string does not fail, but it is desired to check that in the presence of a prime string fault condition, the online string rolled back appropriately and did not get ahead of the prime string and (b) the prime string does fail and the online string becomes prime and must take over using the critical sequence that had been in place on the old prime string before the fault occurred. Point (a) is particularly important to check because fault protection does not run in the online string. This asymmetric behavior can cause the two strings to get out of phase. [1]

Having noted that it may be necessary for the prime and online strings to exchange their identities should a serious error occur in the prime string, a method to validate this requirement is needed. This point as addressed in the next section where we develop the LTL formula for this process.

A more abstract way to validate the over-all effect of the synchronization requirement on the buffers in the prime and the online strings is to check that aged mark points are always in agreement with each other. This condition can be stated by using the safety condition that the aged mark point p in the prime string never disagree with the aged broadcast mark point q in the online string. The corresponding safety condition would be

$w = \Box(p = q)$

meaning that $w$ should always be false.

## 8.2 Case 4 Swapping Strings

If a prime central interfering fault occurs in the prime string, it will be necessary for the online string to transition to prime status; take control of the system bus; roll back to the most recent valid mark point; and begin execution of the critical sequence. Subsequently, having repaired the fault, the old prime string will notice that there is already an existing prime string and transition to online status. This last point assumes that the fault in the old prime system is repairable. All of the above will be carried out by a correctly executing model.

To validate the model's behavior, a Büchi automaton is added to the system that captures this behavior. To construct this FSM, let a, b, and

[1]Normally a communication protocol for a two element system is validated symmetrically. Each of two communication elements function as both a transmitter or receiver during normal operations. There is no functionality in one component that is not in the other. Consequently, validation of one element suffices to validate the system. Here each of the two strings (elements) can assume the other strings' identity. This happens mechanically but not functionally. Functionally, each string contains an operational online string machine or an operational prime string machine with different characteristics. This means the entire string functions as either an online or as a prime string, not as both. The distinction being that the online string does not have the fault protection funtionality of the prime string; it does not control the spacecraft; and it always functions as a receiver of prime string information for synchronization purposes. Accordingly, the validation of the entire system must treat both strings. One string can not be validated in isolation and constitute the validation of the other one.

| Description | a (platform a) | b (platform b) | c (fault) |
|---|---|---|---|
| True | prime | prime | has occurred |
| False | online | online | no fault |

Table 2: Platform a and b Parameters

c be Booleans representing GDRS components according to Table 2. The LTL for this case is derived as follows.

When platform a is running the prime string, platform b is online ($\neg b$); fault c occurs in platform a, now or in the future we demand that platform b transition to prime status; platform a transitions to on line status ($\neg a$); and that the fault c be repaired ($\neg c$). It must always be the case that this series of events take place. The liveness condition expressing this sequence of events is:

$f_1 = \Box(p_p \rightarrow \Diamond q_p)$

where $p_p = a \wedge \neg b \wedge c$

$\mathrm{q}_p = b \wedge \neg a \wedge \neg c$

It could also be the case that the roles of each of the platforms might have been reversed at the outset with platform b running the prime string and with fault c occurring on platform b. Then now or in the future we demand that platform a transition to prime status; platform b transition to online status ($\neg b$); and fault c be repaired ($\neg c$). The liveness formula expressing this condition is

$f_2 = \Box(r_p \rightarrow \Diamond s_p)$

where

$r_p = b \wedge \neg a \wedge c$ and

$s_p = a \wedge \neg b \wedge \neg c$

Again using equation 1, both of these conditions can be combined to produce 1 liveness condition:

$f = \wedge_{i=1}^{2} f_i = \Box(p_p \rightarrow \Diamond q_p) \wedge \Box(r_p \rightarrow \Diamond s_p)$ (5)

Again, as in the derivation of equation **??**, we want to demand that we know in each case that a fault did occur, each element of equation 8.2 must be preceded by $\Diamond p_p$ amd and $\Diamond r_p$ giving

$f = \wedge_{i=1}^{2} f_i = \Diamond p_p \wedge \Box(p_p \rightarrow \Diamond q_p) \wedge \Diamond r_p \wedge \Box(r_p \rightarrow \Diamond s_p)$ (6)

This formula allows both strings to swap states multiple times during the running of a single critical sequence should that be necessary. The corresponding Büchi automaton is generated from this expression using the SPIN LTL conversion option. Using the same reasoning implied by equation 1 all of the liveness results derived so far can be combined to yield:

$f = \wedge_{i=1}^{8} f_i = \Diamond p Wedge \Box(p \rightarrow \Diamond q) \wedge \Diamond r \wedge \Box(r \rightarrow \Diamond s) \wedge \Diamond \wedge \Box(u \rightarrow \Diamond v) \wedge \diamond u \wedge \Box(u \rightarrow \Diamond v) \wedge \Diamond h \wedge \Box(h \rightarrow \Diamond i) \wedge$

$\Diamond j \wedge \Box(j \rightarrow \Diamond k) \wedge \Diamond l \wedge \Box(l \rightarrow \Diamond m) \wedge \Diamond p_p \wedge \Box(p_p \rightarrow \Diamond a_p) \wedge \Diamond r_p \wedge \Box(r_p \rightarrow \Diamond s_p)$

Using the distributive property of the *henceforth operator* $\Box$, with respect to *logical and* $\wedge$, this formula simplifies to

$f = \wedge_{i=1}^{8} f_i = \Box(\Diamond p \wedge (p \rightarrow \Diamond q) \wedge \Diamond r \wedge (r \rightarrow \Diamond s) \wedge \Diamond u \wedge (u \rightarrow \Diamond v) \wedge \diamond h \wedge (h \rightarrow \Diamond i) \wedge$

$\Diamond j \wedge (j \rightarrow \Diamond k) \wedge \Diamond \wedge (l \rightarrow \Diamond m) \wedge \Diamond \wedge (p_p \rightarrow \Diamond a_p) \wedge \Diamond \wedge (r_p \rightarrow \Diamond s_p))$

Again the Büchi automaton generated from this compound liveness condition would be a part of the model represented by:

$P \times A_{\neg f}$

The executing validation system will now routinely check the automaton to see that the rollback takes place correctly on either platform. It also checks that the platforms swap their software identities when a fault of type c occurs.

A liveness property demanding that a running critical sequence terminate at some point; that once a fault occurs, it will eventually be repaired; and that certain prohibited combinations of the protocol flags never occur (safety property) were also included in the validation run.

## 9 Validation Using the PROMELA/SPIN System

The states referred to here are those found in Figures 5 and 6 of the Statecharts model of the GDRS.

The SPIN system is based on the PROMELA language which is an extension of a smaller language named Argos that was developed in 1983 for protocol validation, by Holzmann[4]. It differs from other guarded command languages in that the statements are not aborted when all guards are false but they block, thus providing the required synchronization. As can be seen by inspection, the PROMELA code in the example that follows is based on the C language.

The two major non-deterministic language constructs of PROMELA are the selection structure and the repetition structure. The selection structure randomly selects one of its elements for execution. The selection structure:

```
if
:: (a! = b) - > option1
:: (a == b) - > option2
fi;
```

selects either the first choice followed by the :: or the second one. If the first option is selected and the guard statement $(a! = b) - >$, is true, option1 is executed. If all guard statements are false, the execution blocks until the condition becomes true. A similar result holds for the second selection.

The repetition structure repeatedly chooses from the elements within its boundary. Otherwise it behaves in the same fashion as the selection structure, blocking when all of the guards are false. As an illustration the repetition structure analogous to the selection structure above is:

```
xin
        do
        :: (a! = b) - > option1
        :: (a == b) - > option2
        od;
```

By nesting a selection structure inside of a suitably defined repetition structure, the GDRS requirements were modeled down to the design level (interrupt level). This procedure imposes an ordering upon the actions that take place at each of the eight interrupt levels of the system.

Significantly, the structure allows for the following fundamental three step sequence to take place in the prime system. First, in the early part of the interrupt cycle, GDRS data is collected for transmission to the online system in the following second. Second, during interrupt level 5 data collected in the previous second is broadcast to the online system via rendezvous handshake. Finally, the main data structure from which the next second's broadcast in generated is updated from the present second's data. This is done at interrupt level 7.

This structure was imposed upon upon each substate shown in the Harel Statecharts in Figures 5 and 6. The use of this structure has the advantage that when jumping from one state to another at a certain interrupt level, the timing is preserved across states. The new selection structure at the destination state places control at the same interrupt level preserving time ordering across states. This result is illustrated below.

```
#define RTI_0(t == 0)
#define RTI_1(t == 1)
#define RTI_2(t == 2)
#define RTI_3(t == 3)
#define RTI_4(t == 4)
#define RTI_5(t == 5)
#define RTI_6(t == 6)
#define RTI_7(t == 7)

int t;

t = 0;

  Startup:
do
  ::if
    ::RTI_0- >
    ::RTI_1- >
    ::RTI_2- >
    ::RTI_3- > (CrossStringStateVector[2/_pid] == 0); break- >
    ::RTI_4- >
    ::RTI_5- >
    ::RTI_6- >
    ::RTI_7- >
  fi;
  atomic{t + +- > t = t%8}- >
od;

RAM_INIT:
do
  ::if
    :: RTI_0 >
    :: RTI_1- >
    :: RTI_2- >
    :: RTI_3- >
    :: RTI_4- >
    :: RTI_5- >
    :: RTI_6- >
    :: RTI_7- >
  fi;
  atomic{t + +- > t = t%8}- >
```

```
od;
```

In state Startup if the cross-string-state vector for the other string has a value of 0, during interrupt 3, the guard condition is true and the break command would cause the state to transition to state RAM_INIT, entering it during interrupt 3.

The following slightly more complex example illustrates the method for implementation of the PROMELA model from the Statechart model of the GDRS shown in Figure 5. Consider the state labeled SFP Active. The PROMELA code for this state was written as follows:

```
SFP_Active :
do
::if
  ::RTI_0-> State[_pid] = 17;
     if
       :: SFP = 0->
          SfpFault = 0->
       ::skip->
     fi;
  ::RTI_1->
  ::RTI_2->
  ::RTI_3->
  ::RTI_4->
  ::RTI_5-> if/* was a peripheral interfering fault generated? */
       :: FLT?[sfpfault(SfpFault0)]-> FLT?sfpfault(SfpFault0)
       ::else->
     fi;
       BUS!bus(CS, CM, SFP, Mark, SfpFault);
   ::RTI_6->
   ::RTI_7->
     if
       ::!SfpFault && !SFP->
          goto PRIME_Fault_idle ->
                    ::else->
     fi;
 fi;
 atomic{t++-> t = t%8}->
od;
```

Here, at RTI_0 the CrossStringStateVector for the prime string is set to 17. Next the selection structure nondeterministically selects one of the two cases. The first option cancels the SFP fault response flag and it turns off the system flag that generated the particular type of fault. In this case we are considering interfering peripheral faults. The second option does nothing meaning that processing in the rest of the system is allowed to go on as it might be possible under the circumstances. If the fault is cleared by the first option, then this result is available for transmission over the GDRS bus during RTI_5. However, it is possible that another fault will be generated entering through the FLT channel. The receipt of this fault is also simulated

RTI_5. That statement is handled with an ordinary deterministic if statement testing to see first if there is a message in the queue on message channel FLT. If there is, this message will be broadcast and seen by both the prime and the online strings. If there is no new fault generated, and the nondeterministic choice above turned off both flags, then this information would be broadcast instead during RTI_5. Then upon reaching RTI_7, control is transferred to the Fault Idle State. This is so since superstate SEQUENCE IDLE is entered at its boundary and the default entry point is Fault_Idle. In the PROMELA code it is called PRIME_Fault_idle to distinguish it from the similar state in the online system.

## 10 GDRS Validation Results

The formal SPIN model was coded in PROMELA and executed using SPIN in the validation mode. Initially, the model itself was validated in the following way. The system was first tested exhaustively without error scenarios. This was carried out using "never" clauses to check that the STB was following the time sequence as stated in the requirements and that the mark points were being properly aged. This process was carried out by asking the prime and online processors to execute a critical sequence that was generated to cover cases of interest.

The test sequence had 13 elements with instructions to be executed in each location with the exception of mark points and at the end of the program. Critical sequences are defined such that the BEGIN_PROG instruction is an implicit mark point; the END_PROG instruction defines the end of the critical sequence and is not considered a mark point. This then ignores the requirement that prohibits instructions immediately preceding or following mark points. However, for purposes of checking the operation of the underlying system that executes the sequences themselves, that requirement was regarded as a constraint on the writers of the critical sequence and the assumption was made is that it would not affect the checking of the rollback requirements.

The sequence that was executed was then additionally chosen to make it as short as possible to keep the search space small. The additional constraint was that all possible combinations of rollbacks could be tried in the presence of faults. The cases of interest are those applicable to Cases 1 through 5 as developed in Section 8.1. The sequence chosen for execution by the model is shown below:

```
cs[1] = BEGIN_PROG;
cs[2] = 2;
cs[3] = 3;
cs[4] = 4;
cs[5] = 5;
cs[6] = MARK + 6;
cs[7] = 7;
cs[8] = 8;
cs[9] = MARK + 9;
cs[10] = 10;
cs[11] = 11;
cs[12] = 12;
cs[13] = END_PROG;
```

This test sequence allows for testing all the error scenarios that could result. Faults were injected into the system leading to Case I scenarios. Inspection of the sequence shows that system performance will be checked when up to 4 seconds are allowed to elapse between mark points. Sequences that have much more than 4 seconds between mark points are not expected to perform any differently under fault scenarios than those with 4 seconds between mark points as far as rollback performance goes. It isn't expected that any real sequence would be used that could have mark points less than 2 seconds apart due to the due to the two second constraint mentioned above.

Because we were able to design the model such that the overall state space vector for the model was very small, we were able to exhaustively validate the model. The fault injection scheme assumed a single fault and recovery scheme. Multiple faults were not considered. The procedure used was to inject one fault into the executing critical sequence in all possible ways that could occur based upon our construction of the model. Then using the LTL schemes developed in section 8.1 checking to see that those conditions were satisfied. This means that the validator will inject a single fault at all possible points in the execution of the sequence. The corresponding rollback behavior is then checked in each instance. The algorithm works as outlined in section 2 and as described in detail by Holzmann [5] and the references cited there. The composite system model checks to see that the rollback is done so that the sequence could be successfully restarted in both the prime and the online systems.

Three suspect types of behavior were identified and are described below. The first two are potential error scenarios. The third is believed to be a discrepancy in the design requirements that could allow for erroneous behavior of the implemented system. All of the behaviors described used spatial synchronization only. That is, both the prime and the online strings were synchronized initially by forcing them to begin execution using a program counter value of 1 ( = BEGIN_PROG) and thereafter used the rendezvous handshake only to maintain synchronization.

## 10.1 Potential Error Scenario One

Depending on how error detection and repair is handled, it may be possible for the prime system to detect and to repair an intermitent error within one second, and then consequently not to broadcast this state to the online system. This would mean that the online sytem would not receive notice of the fault; therefore, it would continue executing its copy of the sequence. Repeated occurrence of this scenario would cause the online system to get way ahead of the prime string, possibly to the point where the online string would complete execution of its copy of the sequence. A subsequent prime failure would then leave the online system without a rollback point. What occurs in this case is that the synchronization functions as before on one second boundaries so that the two systems were still using the STB handshake in RTI_5 but the online system is now one second ahead of the prime string. This problem was noticed in simulation mode and could be easily fixed if present by arranging the order of processing somewhat differently. Figure 7 shows one sequence of events that this scenario would follow. Inspection of the figure shows that repeated injection of this type of intermitent fault effectively stalls the prime execution while the online system will eventually complete executing its copy of the critical sequence.

## 10.2 Potential Error Scenario Two

The next case also would depend upon how faults are handled at the end of the sequence. If a fault occurs in the prime string within two seconds after the end of the program is reached, it was not clear how the rollback procedure would be handled. In the model, the online string would have already completed its execution of the sequence. Accordingly, if the fault were to bring the prime system down, the online string's behavior would be undetermined since the requirements don't specify system behavior in this case. With this issue resolved, this situation could be validated in Cases 4 and 5 which deal with prime failures. Also another anciliary point to check might be what happens if a fault occurs 3 seconds after the sequence terminates in the prime string. Is the roll back process no longer in effect at this point? This again raises issues concerning the treatment of the END_PROG instruction. That is, since the END_PROG instruction is not considered a mark point, it was not known how to handle that scenario.

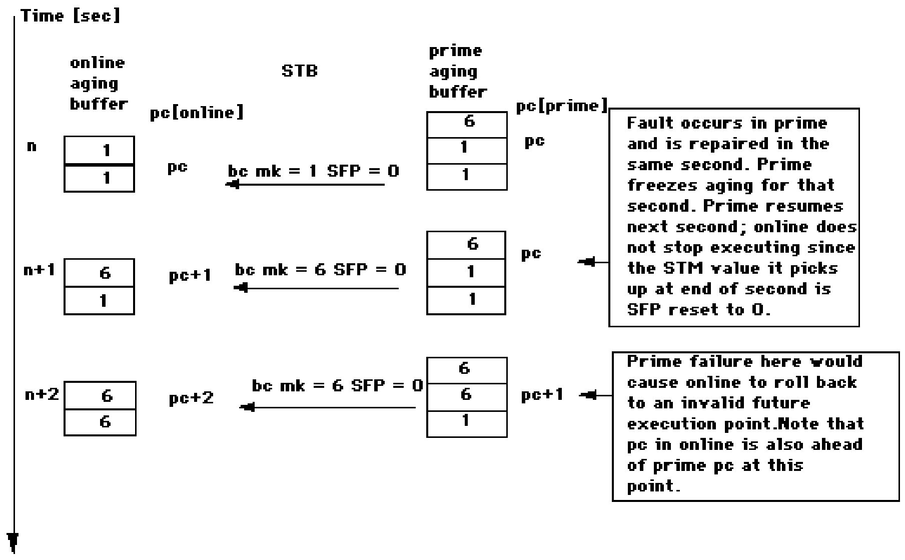
Time [sec]
online aging buffer
STB
prime aging buffer
pc[online]
pc[prime]
n
n+1
n+2
pc
pc+1
pc+2
bc mk = 1 SFP = 0
bc mk = 6 SFP = 0
bc mk = 6 SFP = 0
pc
pc
pc+1
Fault occurs in prime and is repaired in the same second. Prime freezes aging for that second. Prime resumes next second; online does not stop executing since the STM value it picks up at end of second is SFP reset to 0.
Prime failure here would cause online to roll back to an invalid future execution point.Note that pc in online is also ahead of prime pc at this point.

Figure 7: GDRS Prime Fault Repaired in One Second

Figure 8 shows what this scenario looks like. For the sequence used, the aging buffers were assumed to retain the values they contained at the end of execution of the model sequence.

## 10.3 Fault 2 Seconds After Mark Point

This situation is shown in Figure 8. Here the fault occurs 2 seconds after the mark point is encountered in the prime string. The prime system then freezes the aging function giving the aging buffer snapshot shown on the right of the figure at n + 2 seconds. Since the STB broadcast uses the previous second's value for the flags, SFP = 0 is broadcast, the online system continues to execute and ages its mark point by one second giving the configuration shown on the left in the figure at n + 3 seconds. At this point the online system receives the SFP = 1 value and now both agers are frozen. When the fault is subsequently repaired, the prime system will roll back correctly, but the online system will roll to a different point as shown at second m on the left in the figure. This would not cause a problem if the prime system completes the critical sequence. However, if the online system should subsequently have to take over due to a prime failure - possible associated with the [symtomatic] peripheral interfering fault that was just processed, it could roll to an inappropriate block of code. Using a comparison of the internal and the external (broadcast) agers in the online system would not seem to help since the broadcast ager is deemed to be the more reliable source. Also, this problem would not go away if the aging buffers were made deeper or shallower. It would just occur at a different place since it is a consequence of the relative time difference between the two aging schemes.

## 10.4 Summary and Future Work

Table 3 shows a summary of key parameters derived from the output of the SPIN validator. It shows that the state space for the system as described was sufficiently small so that all validations were accomplished in about 30 seconds. In fact the state vector for the validation after optimizing the model was only 92 bytes. Accordingly, it would be quite feasible to add the additional complexity to the model required to handle cases 2 through 5 including the use of a time variable to provide synchronization as per the requirements. As noted earlier any property expressible in the linear temporal logic can be checked using this scheme with the validator. Additionally, non-progress states, end states, assertion violations, and various other

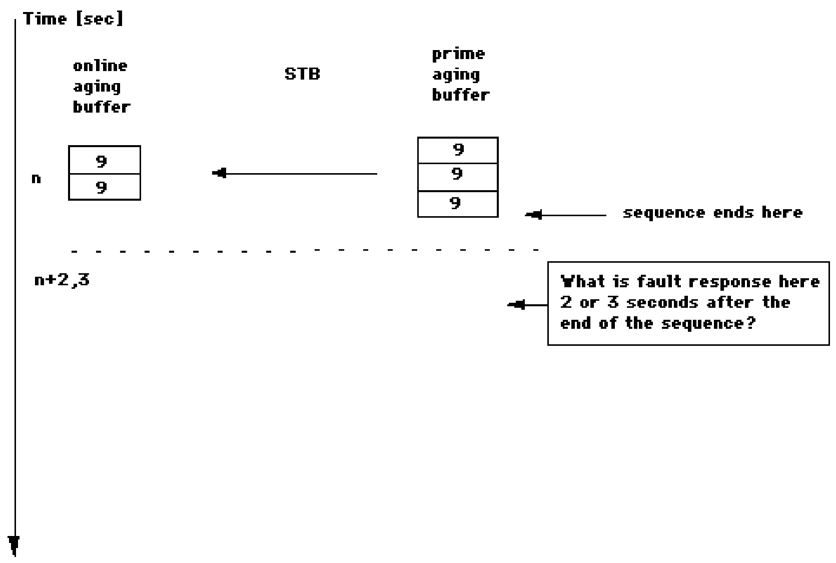

Figure 8: GDRS Fault Two or Three Seconds After Sequence Completion

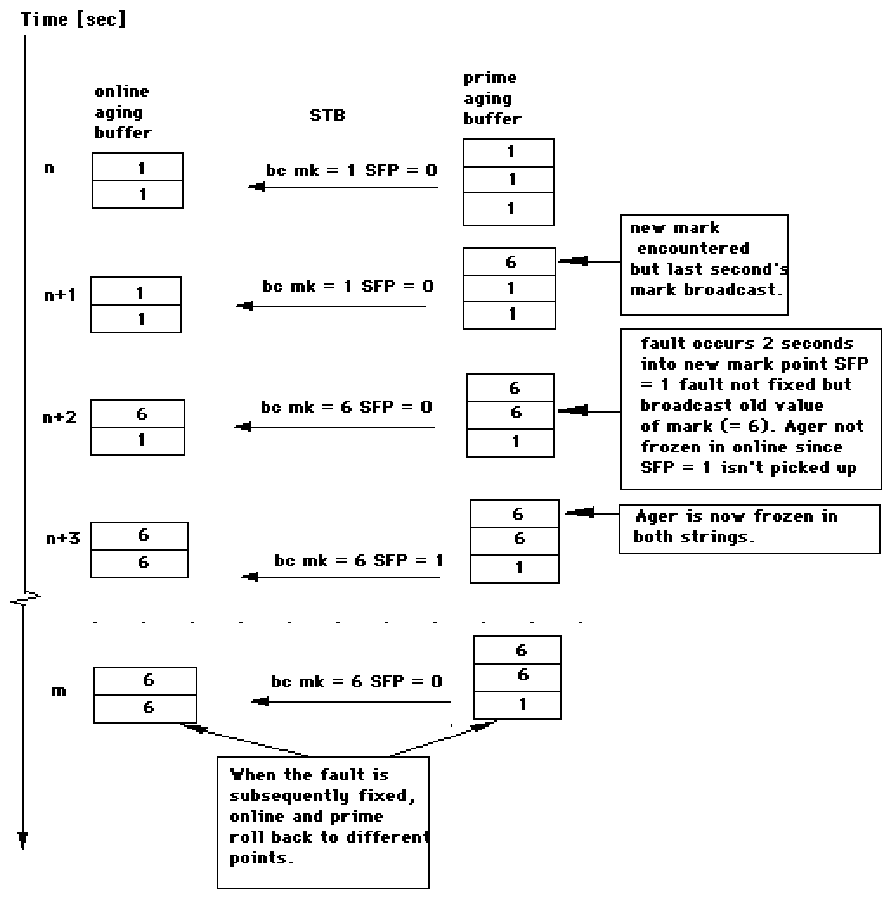
Time [sec]
online aging buffer
STB
prime aging buffer
n
n+1
n+2
n+3
m
bc mk = 1 SFP = 0
bc mk = 1 SFP = 0
bc mk = 6 SFP = 0
bc mk = 6 SFP = 1
bc mk = 6 SFP = 0
new mark encountered but last second's mark broadcast.
fault occurs 2 seconds into new mark point SFP = 1 fault not fixed but broadcast old value of mark (= 6). Ager not frozen in online since SFP = 1 isn't picked up
Ager is now frozen in both strings.
When the fault is subsequently fixed, online and prime roll back to different points.

Figure 9: GDRS Prime Fault Two Seconds After Mark Point

| Property | Description |
|---|---|
| Size of High Level Description in SPIN | Approx 650 lines |
| Assumptions About Faults | Behaviors were checked extensively for the proper handling of a single fault, occurring randomly during a critical sequence execution. |
| Number of Reachable States | Approx. 100, 000 |
| Exhaustive Verification Time of Behaviors | Approximately 30 Seconds per property checked |
| Properties Formalized | 6 |
| Number of These Failing in Verification | 3 |

Table 3: Validation Summary

safety properties can also be checked exhaustively with the validator. And, as mentioned earlier the error trail files generated from the three potential fault scenarios can be annotated and used to test the implementation.

Table 4 provides a brief summary of the potential errors as discussed in the text.

## 11 Future Work

The GDRS validation is a work in progress. Having shown the use of the validation of the model in this initial phase of the work, we plan to complete the other 4 cases as well. Although they are more complex, they should still be quite manageable and we look forward to reporting on them as well in the near future.

Additionally, performance requirements could be validated if the system were modeled with RTSPIN [7]. Using real time operators, the high level design would also validated down to the interrupt level as was done here. As shown, the system uses eight active interrupt levels. Those that contribute something significant in the way of activities that must take place each second would then contribute to the overall validation of the measurable system performance requirements refelcting the activities to keep the two active platforms in sync.

| Potential Error | Description |
|---|---|
| 1. | Occurrence and repair of a fault in the prime both within 1 second may not be communicated to the online and if so would lead to loss of synchronization |
| 2. | Suspect case – should check if the occurrence of a fault at within 2 seconds after the end of program is reached is rolled back and if the occurrence of a fault after 3 seconds is no longer rolled back to the last mark point |
| 3. | Occurrence of a fault at 2 seconds after a mark mark point may lead the online system to roll back to a later markpoint than the prime system |

Table 4: Synopsis of Potential Errors

## 12 GDRS Conclusions

The validation scheme reported on here is efficient because it couples searching of the model state space with specific searches for formal dynamic (liveness) and static (safety) properties of the model's requirements. This procedure is capable of proving requirements completeness and consistency in the model providing the state space can be thoroughly searched. If the state space can not be thoroughly searched it can still provide a high degree of confidence in the degree to which the model is error free. In this later case errors in the model may go undetected. Other errors in the model may go undetected because they were not coded for or because the model does not represent the system accurately. By alternately validating the model against the implementation during the model building process this last problem would be minimized. The discovery of interesting system properties in the model can be used to validate the implementation and to learn more about the behavior of the system itself.

The cost benefit ratio can be very high for testing in this way since once the modeling scheme is constructed, thousands of states can be rapidly searched in the model to select and focus on a significantly smaller number of problem areas. Accordingly, the payback on risk reduction should be high through increased reliability and availability due to (a) detecting any requirements that may be incomplete or inconsistent, (b) discovery of new (unknown) system properties, (c) discovery of possible operations problem areas and (d) suggesting design improvements.

## References

[1] J. R. Buchi. On a decision method in restricted second-order arithmetic. Proceedings of the International Conference on Logic Methodology and Philosophy of Sciences 1960, Stanford University Press, Stanford California, 1960.

[2] D. Harel. Statecharts: A visual approach to complex systems. Technical Report CS84-05, The Weizmann Institute of Science, 1984.

[3] D. Harel and A. Pnueli. *Logics and Models for Concurrent Systems.* Springer, 1985.

[4] Gerard Holzmann. Tracing protocols. *AT&T Technical Journal*, 64:2413–2434, December 1985.

[5] G.J. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall, 1991.

[6] Parameswaran Ramanathan and Kang G. Shin. Use of common time base checkpointing and rollback recovery in a distributed system. *IEEE Transactions on Software Engineering*, 19(6):571–582, June 1993.

[7] S. Tripakis and C. Courcoubetis. Extending promela and spin for real time. Proceedings of the First SPIN Workshop, J-Ch. Gregorie, Ed. INRS-Telecommunications, Montreal, QC, 1995.